2. Basic Organization of a Computer

2.1 The block diagram

Most of the computers available today on the market are the so called **von Neumann computers**, simply because their main building parts, CPU or processor, memory, and I/O are interconnected the way von Neumann suggested. Figure 2.1 presents the basic building blocks of today's computers; even though there are many variations, and the level at which these blocks can be found is different, sometimes at the system level, other times at board level or even at chip level, their meaning is yet the same.

- **CPU** is the core of the computer: all *computation* is done here and the whole

- system is *controlled* by the CPU

- the program and the data for the program are stored in the **memory**

- **I/O** provide means of entering the program and data into the system. It also allows the user to get the results of the computation.

2.2 Computation and control in CPU

The computation part of the CPU, called the datapath, consists of the following units:

- **ALU** (Arithmetic and Logic Unit) which performs arithmetic and logic operations;
- **registers** to hold variables or intermediary results of computation, as well as special purpose registers;
- the **interconnections** between them (internal buses).

The datapath contains most of the CPU's **state**; this is the information the programmer has to save when the program is suspended; restoring this information makes the computation look like nothing had happened.

The state includes the user visible general purpose registers, as well as the **Program Counter** (PC: it contains the address of the next instruction to be executed), the **Interrupt Address Register** (IAR: contains the address of the instruction being suspended), and a **Program Status Register** (PSR: this usually holds the status flags for the machine, like condition codes, masks for interrupts, etc.).

With a few exceptions (like PC or IAR) there is no rule to indicate if some special signification register must be kept in the general purpose area (also called the **register-file**), or in a specially dedicated register. Should the stack-pointer or the frame-pointer, for instance, have special registers with dedicated hardware to help them perform the functions they are meant to, or they can simply reside in the register-file?

On one hand a structure without “special features” is “cleaner”, in the sense that it is easier to design and debug; on the other hand there are strong reasons to have special purpose registers, and the most important is *efficiency.* The PC, for example, is a special register, because it has a special function which could be otherwise impossible to perform: its content has to be incremented in each instruction with some value; special hardware helps optimizing this function; as a matter of fact, in many designs, the program counter is closer to a counter than to a simple D-type register.

Specialized hardware also means that some functions in the machine may execute in a parallel fashion, thus increasing the efficiency: using the same example, the program counter can be incremented while some register(s) in the register-file are read/written, and maybe a memory access is in progress.

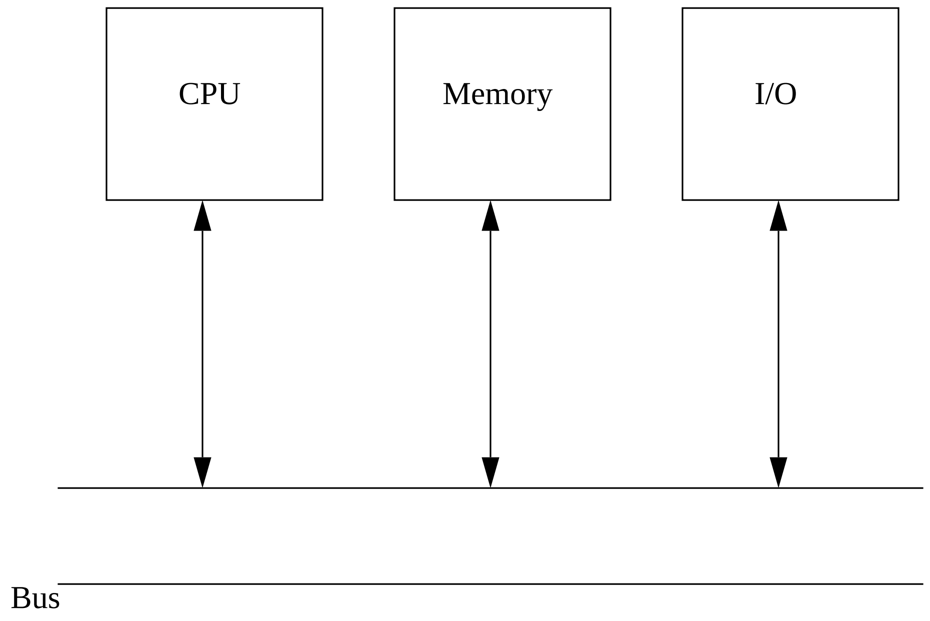

FIGURE 2.1 The building blocks of a computer.

It is also to be mentioned that some special registers can be accessed only by specialized instructions (in the case of PC only by jumps, call/return, branches, with all their variants), thus providing superior protection against accidental alteration, as compared with a general purpose register.

It can be long argued about what functions the ALU should perform, and there are at least two aspects to be considered:

- **encoding**: the operation to be performed in the CPU is somewhere encoded in the instruction, using a number of bits; with n bits one can specify 2n different binary configurations, i.e. that many ALU operations. If n is too small then it will be impossible to accommodate the minimum number of functions the ALU should perform; if the designer is too greedy then fewer bits will remain available to encode other information in the instruction (as for instance, where are the operands to be used, etc.), not to mention the explosive increase in the ALU's complexity. For the time being, three or four bits seem to be enough as control lines for the ALU.

- **functionality**: which is the best set of operations to implement, while keeping the design at reasonable dimensions, and, in the mean time without impairing the programmer's ability to implement any function from the basic set of functions we provide.

Example 2.1 IMPLEMENTATION OF OPERATIONS:

Assume that the instruction set has instructions with the following formats:

```
operation  destination, operand1, operand2
```
or
```
operation   destination, operand
```

where operation specifies what is to be performed with the operands operand1 and operand2, or with operand, and destination is the place where the result is to be stored. Suppose also that the only logic instructions are AND, OR, NOT. Show how to implement the XOR operation; the operands are in registers r1 and r2.

Answer: Use the relation:

$$A \text{ xor } B = (\bar{A} \text{ and } B) \text{ or } (A \text{ and } \bar{B})$$

The following sequence of code implements the XOR:

```
xor:  NOT   r3, r1          # the complement of A in r3
      AND   r3, r2, r3      # the first and
      NOT   r2, r2          # the complement of B in r2
      AND   r2, r1, r2      # the second and
      OR    r3, r2, r3      # final result in r3
```

Now let's consider another example in which the logic operations available are different from those in example 2.1.

Example 2.2 IMPLEMENTATION OF OPERATIONS:

Suppose you have the same instruction formats as in example 2.1, but the only available logic instructions are AND, OR, XOR. Implement the NOT operation; the operand is in register r1 and the content of register r0 is always zero.

Answer: Use the fact that:

$$A \text{ xor } 1 = \bar{A}$$

The following sequence of code implements the NOT operation:

```
not:  SUB   r2, r0, 1       # make all 1's in r2
      XOR   r2, r2, r1      # final result in r2
```

The above sequence of code assumes that subtracting one from zero (integer substraction) yields a all ones result; this is true for unsigned and two's complement representation integers.

Example 2.2 points out that the common case has to be consider when choosing an instruction set; the efforts in design will probably go towards optimizing the common case. Certainly the designer could consider implementing both the NOT and XOR operations in the instruction set (i.e. to have corresponding instructions): same questions emerge again, are there enough opcodes to implement a new operation, and what is the hardware price we have to pay for it? Usually more hardware means a lower clock cycle and the specter of offsetting the overall performance.

It is now the time to discuss about interconnections inside the CPU and to sketch a CPU. Basically the question is *how many internal buses should the CPU have?*

If space/low-price are a must then a single internal bus may be considered as the one in figure 2.2. This approach has however a big drawback: little flexibility in choosing the instruction set; most operations have as an operand the content of the accumulator, and this is also the place where the result goes. Due to its simplicity (simple also means cheap!), this was the solution adopted by the first CPUs.

When we say simple we mean both hardware simplicity and software simplicity: because one operand is always in the accumulator, and the accumulator is also the destination, the instruction encoding is very simple: the instruction must only specify what is the operation to be performed and which is its second operand. Could the designers have encoded more than this in the first 8-bit integrated CPUs (the Intel 8080 or the Zilog Z80, the most popular 8-bit microprocessors, both appeared in the 70s)?

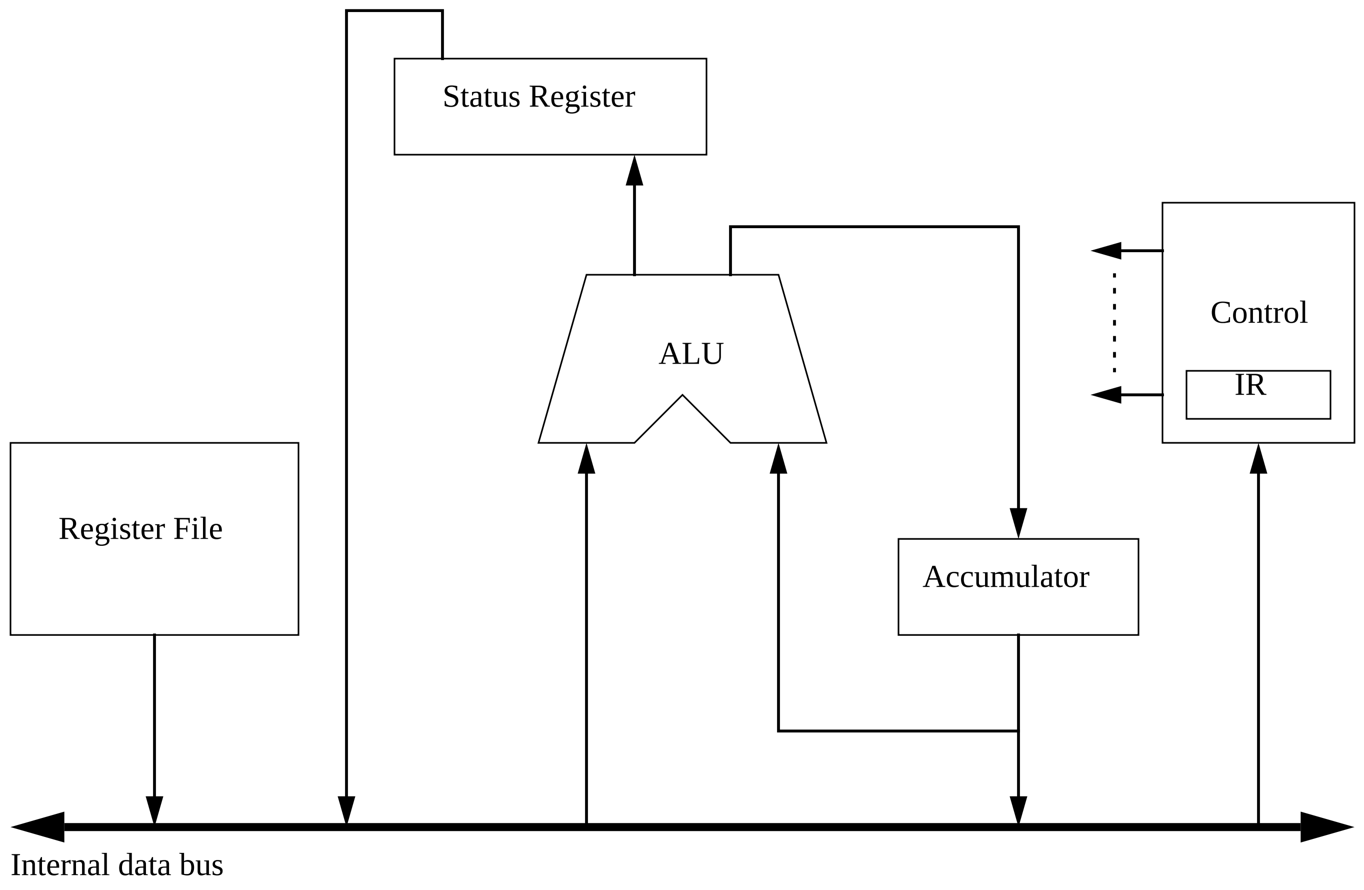

FIGURE 2.2 A possible organization of a CPU, using a single internal data bus. IR is the Instruction Register.

As the technology allowed to move to wider data paths (16, 32, 64 and even larger in the future), it has become also possible to specify more complex instruction formats: more explicit operands, more registers, larger offsets, etc. It is the moment to observe that newer CPU generations are faster due to:

- **faster clock rate** (lower T_{ck}); while the technology features decreased more transistors fit on the same surface and they may operate at higher speed;

- **lower IC**: it takes fewer instructions to perform an integer instruction on 32-bit integers, if the datapath is 32-bit wide as compared with an 8-bit datapath;

- **lower CPI**: with a more involved hardware it is possible to make large transfers (read/store from/to memory in a single clock cycle, instead of several ones as it was the case with narrower datapath CPUs.

Figure 2.3 presents a typical modern CPU, connected to memory. The CPU uses three buses (Op1, Op2 and Dest). The two operands are placed on the two buses, Op1, and Op2, an operation is performed, and the result gets on the Dest bus to be stored in any register connected to the bus.

- **MAR** is the Memory Address Register which holds the memory address during an instruction fetch on a load/store operation;

- **MDR** is the Memory Data Register, used to hold the data to be written into the memory during a store or to temporarily hold the data during a load;

- **temp** is a temporary register used for internal manipulation of data.

Figure 2.3 also assumes that the only way from a register to another is through ALU, therefore ALU must be able to, as one of its functions, pass one operand from input to output.

2.3 Instruction cycle

Obviously there are at least two steps in the cycle of an instruction: fetch (i.e. the instruction is brought into CPU, more precisely into IR) and execute.

At a closer look several substeps can be seen:

1. Instruction fetch step:

$MAR \longleftarrow PC$

$IR \longleftarrow M[MAR]$

The content of PC is transferred into MAR; then the instruction at address MAR is brought into IR.

2. Instruction decode / register fetch step:

Decoding the instruction is the step when the control decides what should be done next; if the instruction has a fixed fields format, then the contents of registers specified in the instruction can be read into A and B at the same time with the decoding.

It is also in this phase that PC has to be updated; how much is to be added to the PC in order to get the new PC (i.e. the address of next instruction to be executed)? Various factors are to be considered, like instruction width, byte/word addressable memory, memory alignment.

3. Execution

In the case of an arithmetic logic operation whose operands are in registers the operation is performed.

If the instruction is a load/store, then the address has to be computed and only then the operation can be performed.

In the case of a branch/jump operation the target address has to be computed and, for a conditional branch/jump, PC may be updated or not depending on the flag (condition) being tested.

The execution step could be further divided into specific substeps for each instruction or class of instructions.

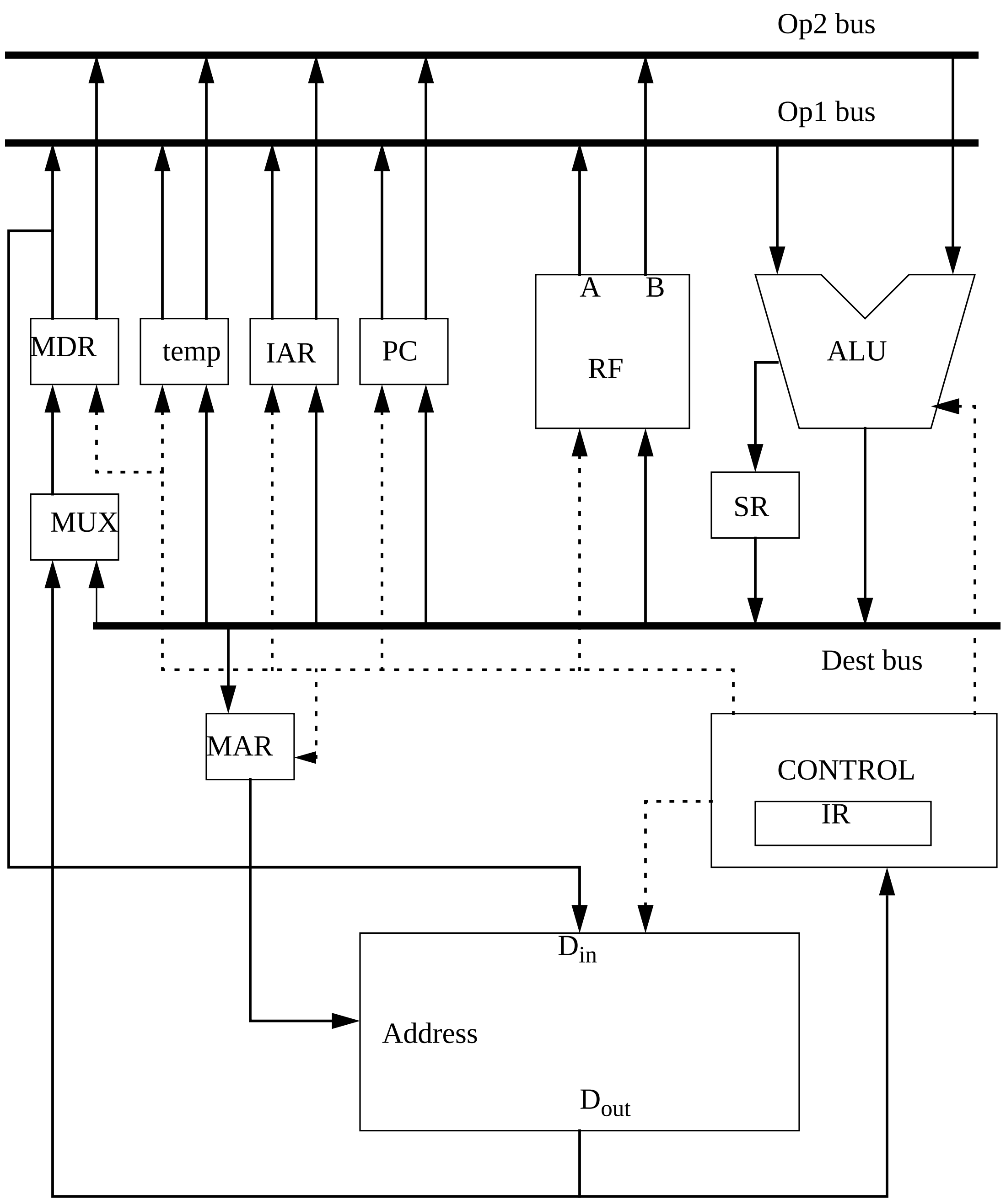

FIGURE 2.3 A typical CPU organization; it is represented in connection with the memory.

Exercises

2.1 Give some arguments for having the stack pointer as a special purpose register, and indicate which is the hardware required to manage it.

2.2 The block diagram in Figure 2.3 represents the Program Counter as a normal register, i.e. without dedicated hardware around it. How can it be incremented, and how long does it take to do so? In which case would PC require dedicated hardware?